```python
from datetime import datetime, timedelta
from dateutil.parser import parse
import random
import pandas as pd
import numpy as np

import sklearn
from sklearn.metrics import silhouette_score
from sklearn import preprocessing

import matplotlib.pyplot as plt
from matplotlib import cm

from pyclustering.cluster.kmeans import kmeans
from pyclustering.utils.metric import type_metric, distance_metric
from pyclustering.utils.metric \
    import euclidean_distance, manhattan_distance, chebyshev_distance,
    minkowski_distance
from pyclustering.cluster.center_initializer import kmeans_plusplus_initializer


class PycClusteringPeople:
    df_corona = None  # initial loaded data
    added_column_list = []  # Columns added by calculation within class

    sse_list = []  # list for SSE of each cluster
    sil_score_list = []  # list for Silhouette score of clustering result
    centroids_coord_list = []  # list for storing coordinates of centroids

    # member variable for custom distance function
    weight_list = []  # weight values list

    # TODO: Discard below 4 values
    num_of_data = 0
    data_cal_count = 0
    num_of_cent = 0
    cent_cal_count = 0

    def __init__(self, file_path):
        self.load_data(file_path)
        self.compute_severity(parse('2020 7 3'))

    def load_data(self, file_path):
        """
        method to load .csv file
        :param file_path: string, the path of file
        :return:
        """
        self.df_corona = pd.read_csv(file_path)

    def compute_severity(self, base_date):
        """
        method to preprocess the data for distance function
        :return: None
        """
        col_num = len(self.df_corona)  # the number of rows from loaded data
        today = datetime.now().date()  # date of today, YEAR-MONTH-DAY

        # selecting specific column to compute 'severity'

        severity_list = []  # list for storing severity result

        for i in range(col_num):
            incur_date_col = self.df_corona['Incurred Date']
            status = self.df_corona['Covid Status']
            severity = 0  # default is healthy. 0.
            if status[i] == 'Contacted':  # contacted person?
```

```
                # formula for contacted person:
                #   x = 1 - ((today's date) - (infected date)) * 0.05)
                elapsed_days = (base_date.date() - parse(incur_date_col[i]).date()).days
                severity = (1 - (elapsed_days * 0.05)) * 0.5

            elif status[i] == 'Confirmed':  # confirmed person?
                # formula for confirmed person:
                #   x = (1 - ((today's date) - (infected date)) * 0.05)) / 2
                elapsed_days = (base_date.date() - parse(incur_date_col[i]).date()).days
                severity = 1 - (elapsed_days * 0.05)

            # add the value to the list
            # and rounding to solve floating-point problems
            severity_list.append(round(severity, 4))
        self.df_corona["Severity"] = severity_list
        self.added_column_list.append("Severity")

    def initialize_random_centroid(self, num_centroid, is_0_1_normalized=True):
        """
         A function that generates a centroid of random coordinates-
         -as many as the number of clusters received.
         :param num_centroid: int, the number of centroids
         :param is_0_1_normalized: boolean, Whether the feature is normalized
         :return: the random coordinates of centroids list
         """
        if is_0_1_normalized:  # are all columns normalized to 0-1?
            # for i in num_centroid:
            return [[random.uniform(0, 1), random.uniform(0, 1)] for _ in range(num_centroid)]

        else:  # there is an unnormalized column
            pass

    def pyc_cluster_kmeans(self,
                           target_col_name_list,
                           num_cluster,
                           weight_list,
                           distance_function):
        """
         function to cluster data
         :param target_col_name_list: list, name list to extract column as feature
         :param num_cluster: int, the number of clusters
         :param weight_list: list, weight list of features
         :param distance_function: string, the abbreviation of distance function
         :return: list, clustered result
         """
        self.weight_list = weight_list

        # my_distance_function = lambda p1, p2: p1[0] + p2[0] + 2
        if distance_function == 'eu':
            # metric = distance_metric(type_metric.EUCLIDEAN)
            metric = euclidean_distance
        elif distance_function == 'ma':
            # metric = distance_metric(type_metric.MANHATTAN)
            metric = manhattan_distance
        elif distance_function == 'mi':
            # metric = distance_metric(type_metric.MINKOWSKI)
            metric = minkowski_distance
        elif distance_function == 'c_eu':
            metric = distance_metric(type_metric.USER_DEFINED, func=self.weighted_euclidean_distance)

        target_data = self.df_corona.loc[:, target_col_name_list]  # To select required data
        if "Age" in target_col_name_list:  # scaling only "Age" column
            age = target_data.loc[:, "Age"].values.reshape(-1, 1)
            scaler = preprocessing.MinMaxScaler()
```

```python
            # data = scaler.fit_transform(data)   # To scale data from 0 to 1
            target_data.loc[:, "Age"] = scaler.fit_transform(age)

        # set the number of data and centroids
        self.data_cal_count = num_cluster*len(target_data)
        self.num_of_data = num_cluster*len(target_data)
        self.cent_cal_count = 1
        self.num_of_cent = 1

        # initializing centroids
        # initial_centers = kmeans_plusplus_initializer(target_data, num_cluster).initialize()
        # initial_centers = self.initialize_random_centroid(num_cluster)
        initial_centers = [[i*0.1, i*0.1] for i in range(num_cluster)]

        kmeans_instance = kmeans(target_data, initial_centers, metric=metric)

        kmeans_instance.process()
        clustered_list = kmeans_instance.get_clusters()
        clustered_list = self.pyc_result_to_column(clustered_list, len(target_data))

        # add the column
        self.df_corona['Cluster ID'] = clustered_list

        # storing the coordinates of centroids
        self.centroids_coord_list.append(kmeans_instance.get_centers())

        # storing SSE(Sum of Squared Errors)
        self.sse_list.append(kmeans_instance.get_total_wce())

        # strong Silhouette Score
        self.sil_score_list.append(silhouette_score(target_data, clustered_list))

        return clustered_list

    def pyc_result_to_column(self, pyc_cluster_result, people_num):
        """
        function to change shape of Pyclustering to pandas
        :param pyc_cluster_result: nd list, result of clustering using Pyclusterin
        :param people_num: int, the number of people(data)
        :return: list, re-shaped list
        """
        clustered_list = [0 for _ in range(people_num)]

        cluster_id = 0
        for id_list_of_a_cluster in pyc_cluster_result:
            for idx in id_list_of_a_cluster:
                clustered_list[idx] = cluster_id
            cluster_id += 1

        return clustered_list

    def weighted_euclidean_distance(self, point1, point2):
        """
        custom distance function
        :param point1: list, list of feature values or coordinates list of centroid
        :param point2: coordinates list of centroid
        :return: distance between point1 and point2
        """
        distance = 0.0  # distance between point1 and point2

        # TODO: condition is should be changed to type comparison
        if self.data_cal_count > 0:  # when calculating the distance of two coordinates
            # point 1 is data.
            # point 2 is centroid
            self.data_cal_count -= 1
            for weight, p1_coord, p2_coord in zip(self.weight_list, point1, point2):
                distance += weight * (p1_coord - p2_coord) ** 2.0

        else:  # when updating Centroid
            # point 1 and 2 are centroid
            self.cent_cal_count -= 1
            if self.cent_cal_count == 0:
                self.data_cal_count = self.num_of_data
                self.cent_cal_count = self.num_of_cent

            for prev_cent, curr_cent in zip(point1, point2):
                for weight, pc_coord, cc_coord in zip(self.weight_list, prev_cent, curr_cent):
                    distance += weight * (pc_coord - cc_coord) ** 2.0

        return distance ** 0.5

    def display_clustering_result(self,
                                  num_cluster,
                                  cluster_idx_list,
                                  cluster_predicted_list):
        """
         function to display clustering result on console as tabular type
         :param num_cluster: int, the number of cluster
         :param cluster_idx_list: list, cluster index list, ie. [2, 3, 4, 5, 6]
         :param cluster_predicted_list:
         :return:
         """

        severity_list = self.df_corona["Severity"].values.tolist()
        age_list = self.df_corona["Age"].values.tolist()
        if type(cluster_predicted_list) != list:
            cluster_predicted_list = cluster_predicted_list.tolist()
        people_num_of_a_cluster_list = []
        avg_age_of_a_cluster_list = []
        avg_severity_of_a_cluster_list = []

        print(f"Number of Clusters: {len(cluster_idx_list)}")

        for cluster_idx in cluster_idx_list:  # 1 cluster
            num_people = cluster_predicted_list.count(cluster_idx)
            id_target_data_tuple_list = []
            target_severity_list = []
            target_age_list = []

            for person_idx in range(len(cluster_predicted_list)):
                if cluster_idx == cluster_predicted_list[person_idx]:
                    target_severity_list.append(severity_list[person_idx])
                    target_age_list.append(age_list[person_idx])
                    id_target_data_tuple_list.append((
                        person_idx+1,  # [0] of tuple is id
                        age_list[person_idx],  # [1] of tuple is age
                        round(severity_list[person_idx], 2)))  # [2] of tuple is severity

            people_num_of_a_cluster_list.append(num_people)

            print(f"\tCluster {cluster_idx}:")
            print(f"\t\tNumber of People: {num_people}")
            # print(f"\t\t\t{'ID':<4}{'Age':<4}{'Severity Value'}")
            # for person_in_cluster in id_target_data_tuple_list:
            #     print(f"\t\t\t{person_in_cluster[0]:<4}"
            #            f"{person_in_cluster[1]:<4}"
            #            f"{person_in_cluster[2]}")
            print(f"\t\tMinimum of Age values: {min(target_age_list)}")
            print(f"\t\tMaximum of Age values: {max(target_age_list)}")
            print(f"\t\tAverage of Age values: "
                  f"{round(sum(target_age_list) / len(id_target_data_tuple_list), 2)}")
            print(f"\t\tMinimum of Severity values: {min(target_severity_list)}")
            print(f"\t\tMaximum of Severity values: {max(target_severity_list)}")
            print(f"\t\tAverage of Severity values: "
                  f"{round(sum(target_severity_list) / len(id_target_data_tuple_list), 2)}")
            print(f"\t\tThe Coordinates of Centroid:")
            coords = self.centroids_coord_list[num_cluster-2][cluster_idx]
            print(f"\t\t\tX1 (Severity): {round(coords[0], 2)}")
            print(f"\t\t\tX2 (Age): {round(coords[1], 2)}")
            try:
                avg_age_of_a_cluster_list.append(
                    round(sum(target_age_list) / len(id_target_data_tuple_list), 2))
            except ZeroDivisionError:
                avg_age_of_a_cluster_list.append(0)

            try:
                avg_severity_of_a_cluster_list.append(
                    round(sum(target_severity_list) / len(id_target_data_tuple_list), 2))
            except ZeroDivisionError:
                avg_severity_of_a_cluster_list.append(0)

            print()  # float 1 line
        self.display_summary_table(people_num_of_a_cluster_list,
                                   avg_age_of_a_cluster_list,
                                   avg_severity_of_a_cluster_list)
        print()  # float 1 line

    def data_to_csv(self, num_cluster):
        """
        function to save data as .csv file
        :param num_cluster: int, the number of cluster.
        :return: None
        """
        temp_df = self.df_corona.__deepcopy__()

        file_name = f"clustered_corona_data_k={num_cluster}_" \
                    f"{'Severity_Age'}_{''.join(str(self.weight_list))}.csv"
        temp_df.to_csv(file_name, encoding='utf-8-sig')

    def display_load_data(self):
        """
        function to display data
        :return: None
        """
        print(f"Total number of People: {len(self.df_corona)}")
        print(f"{'ID':<4}"
              f"{'Age':<4}"
              f"{'Covid Status':<13}"
              f"{'Severity':<9}"
              f"{'Address':<10}")
        for i in range(len(self.df_corona)):
            print(f"{self.df_corona['ID'][i]:<4}"
                  f"{self.df_corona['Age'][i]:<4}"
                  f"{self.df_corona['Covid Status'][i]:<13}"
                  f"{round(self.df_corona['Severity'][i], 3):<9}"
                  f"{self.df_corona['Address'][i].split()[0]:<10}"
                  )
        print()  # float 1 line
        grouped_status = self.df_corona['Severity'].groupby(self.df_corona['Covid Status'])

        print(f"Number of healthy people: {grouped_status.count()['Healthy']}")
        print(f"Number of contacted people: {grouped_status.count()['Contacted']}")
        print(f"Number of confirmed people: {grouped_status.count()['Confirmed']}")

        print(f"Average Severity of contacted people: "
              f"{round(grouped_status.mean()['Contacted'], 2)}")
        print(f"Average Severity of confirmed people: "
              f"{round(grouped_status.mean()['Confirmed'], 2)}")
        print()  # float 1 line

    def plot_data(self, num_cluster):
        """
         To plot result
         :return:
         """
        groups = self.df_corona.groupby("Cluster ID")
        fig, ax = plt.subplots()
        for name, group in groups:
            ax.plot(group.Severity, group.Age, marker='o', linestyle="", label=name)
        ax.legend(fontsize=12)
        plt.title("Result of Clustering (K="+str(num_cluster)+", Weight="+str(self.weight_list)+")")
        plt.xlabel("Severity")
        plt.ylabel("Age")
        # plt.show()
        fig.savefig("./Cluster_Result_Plotting/cluster_result_"+str(num_cluster)+"_"+str(self.weight_list)+".png", dpi=300)
        plt.close()

    def display_summary_table(self,
                              people_num_of_a_cluster_list,
                              avg_age_of_cluster_list,
                              avg_severity_of_cluster_list):
        """
         function to display the data as tabular summary
         :param people_num_of_a_cluster_list: list, the number of people in a cluster
         :param avg_age_of_cluster_list: list,
         :param avg_severity_of_cluster_list:
         :return:
         """
        len_id = 17
        len_p_num = 11
        len_age = 13
        len_sev = 15
        len_sum = len_id+len_p_num+len_age+len_sev

        # top row
        print(f"\t{'-'*(len_sum+11)}")
        print(f"\t{'Cluster ID':>{len_id}} "
              f"| {'# of People':>{len_p_num}} "
              f"| {'Avg. of Ages':>{len_age}} "
              f"| {'Avg. of Severity':>{len_sev}} ")

        # contents of table
        cluster_id = 0
        for people_num, avg_age, avg_sev in zip(people_num_of_a_cluster_list,
                                                avg_age_of_cluster_list,
                                                avg_severity_of_cluster_list):
            print(f"\t{cluster_id:>{len_id}} "
                  f"| {people_num:>{len_p_num}} "
                  f"| {avg_age:>{len_age}} "
                  f"|{avg_sev:>{len_sev}}")
            cluster_id += 1

        print(f"\t{'-'*(len_id+1)}"
              f"|{'-'*(len_p_num+2)}"
              f"|{'-'*(len_age+2)}"
              f"|{'-'*(len_sev+2)}-")

        # bottom row
        print(f"\t{'Total':^{len_id}} | {sum(people_num_of_a_cluster_list):>{len_p_num}} |")
        print(f"\t{'SSE':^{len_id}} | {round(self.sse_list[len(people_num_of_a_cluster_list) - 2], 2):>{len_p_num}} |")
        print(f"\t{'Silhouette Score':>{len_id}} "
              f"| {round(self.sil_score_list[len(people_num_of_a_cluster_list) - 2], 2):>{len_p_num}} |")
        print(f"\t{'-'*(len_sum+11)}")

    def draw_silhouette(self):
        """
        method to draw graph using silhouette scores
        :return: None
        """
        pass

    def draw_graph(self):
        """
        method to draw clustering result
        :return: None
        """
        pass

    def draw_elbow_method(self, sse_list):
        """
        method to draw elbow graph using SSE(Sum of Squares Error)
        :param sse_list: list of SSE
        :return: None
        """
        plt.plot(range(2, 10), sse_list, marker='o')
        plt.xlabel("The Number of Cluster")
        plt.ylabel("SSE")
        plt.show()
```